Санкт-Петербургский государственный университет

Прикладная математика и информатика

Исследования операций и принятие решений в задачах оптимизации, управления и экономики

Носкова Екатерина Эдуардовна

Метод имитации отжига для задачи о расписании с параллельными процессорами

Бакалаврская работа

Научный руководитель:

к. ф.-м. н., доцент Григорьева Н. С.

Рецензент:

к. ф.-м. н., доцент Бухвалова В. В.

Санкт-Петербург

2016

Saint Petersburg State University

Applied Mathematics and Informatics

Operation Research and Decision Making in Optimisation,

Control and Economics Problems.

Noskova Ekaterina

Method of simulated annealing in scheduling

Bachelor's Thesis

Scientific supervisor:

docent Grigorieva N. S.

Reviewer:

docent Buhvalova V. V.

Saint Petersburg

2016

Содержание

Введение

Известно, что общая задача о составлении расписания является NP-полной [1]. Существует два подхода к решению этой задачи: нахождение точного решения (подходит для решения небольших задач или единовременного решения каких-то конкретных задач, в силу своего происхождения требующих точного решения) и поиск приближенного решения (подходит для автоматической обработки большого количества задач). В последнем случае мы хотим уменьшить среднюю ошибку по большому массиву задач, при этом сохранив полиномиальную эффективность. Первый подход реализуется такими алгоритмами как полный перебор или метод ветвей и границ [1]. Второй подход реализуется бесчисленным множеством эвристических алгоритмов, об одном из которых пойдет речь в данной работе.

Здесь мы будем обсуждать эффективность эвристического алгоритма имитации отжига для решения общей задачи о расписании.

Работа поделена на разделы: “Введение”, “Глава 1: Предварительные сведения”, “Глава 2: Метод имитации отжига”, “Глава 3: Статистические результаты применения метода к задаче”, “Заключительные замечания” и “Список литературы”. Наиболее важными из них являются Глава 2, в которой подробно описан алгоритм, большей частью реализованный на псевдокоде, и Глава 3, в которой мы приводим статистический анализ эффективности нашей реализации (про реализацию см. далее).

Описываемый алгоритм реализован нами в виде мультиплатформенной программы на языке программирования C++, исходный код которой опубликован в сети Интернет по адресу:
`https://bitbucket.org/noscode/simulated_annealing_for_general_scheduling_problem/`.

Глава 1

Предварительные сведения

1.1. Задача о расписании

Нам дана система заданий $U = \{u_1, u_2, ..., u_n\}$, на которой задано отношение частичного порядка $\prec$: выражение $u_i \prec u_j$ означает, что выполнение задания u_j может быть начато только после завершения задания u_i. Заданы целочисленные времена исполнения заданий $\{t_i\}_{i=1}^{n}$. Для выполнения заданий имеется m идентичных процессоров. Любое задание может выполняться на любом процессоре, и каждый процессор может выполнять не более одного задания в каждый момент времени. Прерывания выполнения заданий не допускаются.

Требуется составить расписание S выполнения заданий процессорами так, чтобы общее время выполнения было минимальным, то есть найти для каждого задания u_i время начала выполнения задания s_i и номер процессора p_i, на котором оно выполняется, так чтобы величина:

$$t_S = \max\{s_i + t_i \mid u_i \in U\}. \tag{1.1}$$

была минимальной.

1.2. Метод имитации отжига (Simulated annealing)

1.2.1. Физическая мотивировка

У каждого металла есть кристаллическая решетка. Совокупность позиций всех атомов будем называть *состоянием системы*, каждому состоянию соответствует определенный уровень энергии. Цель метода — привести систему в *состояние с наименьшей энергией*.

В ходе «отжига» металл сначала нагревают до некоторой температуры, что заставляет атомы кристаллической решетки покинуть свои позиции. Затем начинается медленное и контролируемое охлаждение. Атомы стремятся попасть в состояние с меньшей энергией, однако, с определенной вероятностью они могут перейти и в состояние с большей. Эта вероятность уменьшается вместе с температурой. Переход в худшее состояние, как ни странно, помогает в итоге отыскать состояние с энергией меньшей, чем начальная. Процесс завершается, когда температура падает до заранее заданного значения.

Алгоритм имитации отжига не гарантирует нахождения минимума функции энергии, однако преимуществом метода является то, что он вытаскивает ее из локальных минимумов.

1.2.2. Обозначения

Введем обозначения:
Пусть $\mathscr{S}$ —- множество всех состояний (решений) нашей задачи.
Пусть $S_i \in \mathscr{S}$ —- состояние на i-м шаге алгоритма, а $T_i \in \mathbb{R}$ - температура на i-м шаге.
Также нам понадобится определить четыре функции:

- $\mathscr{E} : S \to \mathbb{R}$ — функция энергии (то, что оптимизируем).

- $\mathscr{T} : \mathbb{N} \to \mathbb{R}$ — убывающая функция изменения температуры с течением времени.
- $\mathscr{F} : S \to S$ — функция, порождающую новое состояние.
- $\mathscr{P} : \mathbb{R}^2 \to (0, 1)$ — функция вероятности перехода в новое состояние. Если функция энергии нового состояния меньше, чем старого, то вероятность перехода равняется 1. Если больше, то чем больше эта разница, тем меньше вероятность перехода.

Функция $\mathscr{E}$ каждому решению по какому-то правилу ставит в соответствие число. Зависит от конкретной задачи.

Функция $\mathscr{T}$ ставит номеру итерации i в соответствие температуру. Функция определяет как долго будет работать наш алгоритм. Если $\mathscr{T}$ будет линейной функцией, то время работы будет относительно большим. В случае же со степенной функцией, скажем $\mathscr{T}(i) = \frac{t_1}{\alpha^i}$, всё закончится очень быстро, но мы можем застрять в локальном минимуме, так и не добравшись до финиша.

Функция $\mathscr{F}$ на основе предыдущего состояния порождает новое состояние-кандидат s_{new}, в которое система может перейти, а может и отбросить.

Вероятность $\mathscr{P}$ перехода чаще всего задается функцией:

$$\mathscr{P}(\Delta E, T) = \begin{cases} 1, & \text{если } \Delta E \leq 0 \\ e^{\frac{\Delta E}{T}}, & \text{если } \Delta E > 0 \end{cases}. \tag{1.2}$$

1.2.3. Формальная схема метода

На вход даются начальное состояние S_0, начальная температура $T_{\max}$ и конечная температура $T_{\min}$, на которой алгоритм прекратит работу.

На каждой итерации алгоритм вычисляет новое состояние $S_{new} = \mathscr{F}(S_{i-1})$. Если оно лучше (функция энергии $\mathscr{E}$ меньше), то алгоритм переходит в это состояние, если нет, то переходит с некоторой вероятностью $\mathscr{P}(\Delta E, T)$. Затем понижается температура: $T_{i+1} = \mathscr{T}(i)$ — и начинается новая итерация.

На выход идет лучшее состояние за все время работы алгоритма.

SIMULATED-ANNEALING($T_{\max}$, $T_{\min}$, S_0)

$i := 1$
$T_0 := T_{\max}$ ▷ Температура на 0 итерации
$S_{best} := S_0$
while $T_i > T_{\min}$ **do**
    $S_{new} := \mathscr{F}(S_{i-1})$
    $\Delta E := \mathscr{E}(S_{new}) - \mathscr{E}(S_{i-1})$
    if $\mathscr{P}(\Delta E, T_i) \geq random(0, 1)$ **then** ▷ Переходим ли в новое состояние
        $S_i := S_{new}$
    else
        $S_i := S_{i-1}$
    if $\mathscr{E}(S_i) < \mathscr{E}(S_{best})$ **then** ▷ Запоминаем лучшее состояние
        $S_{best} := S_i$
    $i := i + 1$
    $T_{i+1} := \mathscr{T}(i)$ ▷ Понижаем температуру
return S_{best}

Глава 2

Применение метода отжига к задаче о расписании

Пусть у нас есть задача о расписании:

- n — количество заданий,
- m — количество процессоров,
- $U = \{u_i\}_{i=1}^{n}$ — множество заданий,
- $\{t_i\}_{i=1}^{n}$ — целочисленные времена выполнения заданий,
- $G = \langle U, E \rangle$ — ациклический граф, который задает отношение частичного порядка на множестве U: дуга $e = (u_i, u_j)$ принадлежит множеству E дуг графа G тогда и только тогда, когда $u_i \prec u_j$. Предполагаем, что множество вершин графа топологически отсортировано: каждое ребро идет от вершины с меньшим номером к вершине с большим.

Для применения метода отжига, у нас есть:

- $\mathscr{S}$ — множество состояний. Очевидно, что состоянием задачи о расписании будет является любое допустимое расписание S.
- $\mathscr{E}(S) = t_S = \max\{s_i + t_i \mid u_i \in U\}$ — функция энергии, она же длина расписания.
- $T_{min} = 1$ — конечная температура, при достижении которой алгоритм прекращает работу.
- $\mathscr{T}(k) = T_k$ — функция изменения температуры на каждой итерации. В смежных задачах часто используются показательные функции, например:

$$T_k = 0.999^k \cdot T_{max}. \tag{2.1}$$

Но тогда количество итераций растет логарифмически при увеличении начальной температуры T_{max}. Это не всегда удобно.

Поэтому будем рассматривать следующую функцию:

$$T_k = T_{max} - \frac{T_{max} - T_{min}}{f(T_{max})} \cdot k, \tag{2.2}$$

для которой количеством итераций будет: $k_{max} = f(T_{max})$.

Вполне интуитивно, что количество итераций для различных n должно быть разным: чем больше n, тем больше k_{max}. Поэтому сделаем начальную температуру зависимой от n.

В случае показательной функции изменения температуры в качестве начальной температуры возьмем функцию: $T_{max} = \frac{n^2}{1000}$. Она была подобрана по необходимому количеству итераций для среднего случая наших экспериментальных данных.

Во втором случае в качестве $f(T_{max})$ выберем линейную функцию. Методом проб и ошибок были найдены необходимое количество итераций для задач и следующие функции:

$$T_{max} = n,$$

$$f(T_{max}) = 7 \cdot T_{max} + 3000.$$

- $\mathscr{F}$ — функция, порождающая новое состояние. Ее варианты будут описаны далее.
- $\mathscr{P}(\Delta E, T) = \begin{cases} 1, & \text{если } \Delta E \leq 0 \\ e^{\frac{\Delta E}{T}}, & \text{если } \Delta E > 0 \end{cases}$ — функция вероятности перехода в новое состояние.

2.1. Представление расписания

Можем представить наше расписание S как некую *перестановку* π заданий $\{u_{\pi(i)}\}_{i=1}^{n}$, которая определяет порядок постановки заданий на выполнение, и вектор $\{p_i\}_{i=1}^{n}$, где p_i — номер процессора, выполняющего u_i-ое задание. Перестановка должна удовлетворять графу предшествования, то есть если $u_i \prec u_j$, то $\pi(i) < \pi(j)$.

Если это условие выполнено, то мы легко можем вычислить время начала s_i любого задания u_i как максимум из времени освобождения процессора p_i и минимального допустимого времени постановки задания. Допустимость времени определяется предшественниками задания — оно не может начаться, пока они не выполнятся.

Подробнее алгоритм можно посмотреть в псевдокоде:

CALCULATE-START-TIMES()

1: **for** $i := \pi(1) : \pi(n)$ **do**
2: $\quad j := p_i$ ▷ Номер процессора
3: $\quad s1 := \max\{s_k + t_k |\ \pi(k) < \pi(i),\ p_k = j\}$ ▷ Время освобождения процессора
4: $\quad s2 := \max\{s_k + t_k |\ \pi(k) < \pi(i),\ u_k \prec u_i\}$ ▷ Минимальное время для задания
5: $\quad s_i := \max\{s1, s2\}$

Легко понять, что такой способ вычисления времени начала задания строит соответствующее данной перестановке расписание минимальной длины. Поэтому можно пренебречь вектором $\{s_i\}_{i=1}^{n}$ вообще.

Однако в нашем случае, на каждой итерации метода отжига нам необходимо вычислять длину расписания, чтобы сравнивать ее. Для этого придется считать времена начала для заданий, что потребует порядка n^2 итераций. Поэтому для повышения эффективности будем хранить оба представления одновременно. Это позволит немного оптимизировать процесс: надо будет пересчитывать времена не для всех, а только для части заданий. Тогда можно переписать код как:

REFRESH-START-TIMES($from$)

1: **for** $i := \pi(from) : \pi(n)$ **do**
2: $j := p_i$ ▷ Номер процессора
3: $s1 := \max\{s_k + t_k |\ \pi(k) < \pi(i),\ p_k = j\}$ ▷ Время освобождения процессора
4: $s2 := \max\{s_k + t_k |\ \pi(k) < \pi(i),\ u_k \prec u_i\}$ ▷ Минимальное время для задания
5: $s_i := \max\{s1, s2\}$

2.2. Алгоритм генерации начального состояния

Для начала работы метода имитации отжига нам надо получить начальное состояние.

Для этого мы будем использовать жадный алгоритм: на каждой итерации находим готовое к выполнению задание с максимальным временем исполнения и ставим его на менее загруженный процессор.

Функция LONGEST-AVALIABLE-TASK() вычисляет доступное задание с максимальным временем исполнения. Для ее реализации необходимо каждому заданию u_i поставить в соответствие флаг $isDone(i)$, который имеет значение $true$, если задание уже поставлено на какой-то процессор.

LONGEST-AVALIABLE-TASK($\{isDone(i)\}_{i=1}^{n}$)

1: $i := 1$
2: **for** $j := 2 : n$ **do**
▷ Если все предшественники элемента выполнены, то сравниваем длины
3: **if** $isDone(k) = true,\ \forall k :\ u_k \prec u_j$ **then**
4: **if** $t_j > t_i$ **then**
5: $i := j$
6: **return** i

Теперь можем написать псевдокод жадного алгоритма:

GREEDY-ALGORITHM()

1: $isDone(p) := false,\ p = 1..n$
2: $loadProcessors := m$-мерный вектор из 0 ▷ Вектор загруженности процессоров
3: **for** $k := 1 : n$ **do**
4: $i :=$ LONGEST-AVALIABLE-TASK($\{isDone(p)\}_{p=1}^{n}$)
5: $\pi(i) := k$ ▷ Создаем перестановку
6: $j := 1$ ▷ Найдем номер менее загруженного процессора
7: **for** $l := 2 : m$ **do**
8: **if** $loadProcessors(l) < loadProcessors(j)$ **then**
9: $j := l$
10: $p_i := j$ ▷ Ставим задание на процессор
11: $loadProcessors(j) := loadProcessors(j) + t_i$ ▷ Увеличиваем загруженность
12: $isDone(i) := true$ ▷ Задание выполнено

2.3. Операция преобразования расписания

В методе имитации отжига нам также необходимо задать функцию $\mathscr{F}$ перехода в новое состояние.

Для этого рассмотрим операцию преобразования расписания MOVE-TASK(i, p_{new}, π_{new}) — она переносит задание $u_i \in U$ на процессор с номером p_{new} и меняет порядок выполнения задания $\pi i = \pi_{new}$.

При этом на π_{new} накладываются ограничения, так как перестановка должна удовлетворять графу предшествования:

$$\max\{\pi(j) \mid u_j \prec u_i\} < \pi_{new} < \min\{\pi(j) \mid u_i \prec u_j\} \tag{2.3}$$

MOVE-TASK(i, p_{new}, π_{new})

```
1: p_i := p_new                          ▷ Поменяли привязку задания к процессору
2: if π_new < π(i) then                  ▷ Изменяем перестановку заданий
3:     for j := π(i) : π_new do
4:         π(j) := π(j) + 1
5: else
6:     for j := π_new : π(i) do
7:         π(j) := π(j) − 1
8: π(i) := p_new
```

Заметим, что если новое расписание, полученное этой операцией, является допустимым, то операция обратима.

Теорема 2.3.1. Если S и $\bar{S}$ - два корректных расписания, то существует конечная цепочка промежуточных **корректных** расписаний таких, что каждое получено из предыдущего применением операции MOVE-TASK $K \leq 2n$ раз.

Доказательство довольно тривиально, оно подробно изложено в [3]. □

2.4. Функции перехода в новое состояние

2.4.1. Первая функция перехода в новое состояние

Теперь мы можем построить функцию, которая будет возвращать нам новое близкое состояние системы.

Будем генерировать номер задания, номер процессора, новый номер перестановки и применять операцию MOVE-TASK. Номера задания и процессора могут быть любыми, ограничиваются только количеством каждого, а вот номер перестановки должен удовлетворять требованию 2.3. Поэтому сначала нужно найти нижнюю и верхнюю границы и затем сгенерировать число между ними.

GENERATE-NEW-POSITION(i)

1: $lowerBound := 0$
2: $upperBound := n + 1$
3: **for** $j := 1 : n$ **do** ▷ Считаем точные границы
4: **if** $u_j \prec u_i$ **then**
5: $lowerBound := \max\{lowerBound; \pi j\}$
6: **if** $u_i \prec u_j$ **then**
7: $upperBound := \min\{upperBound; \pi j\}$
▷ Генерируем число между двумя границами
8: $\pi_{new} := random(lowerbound + 1; upperBound - 1)$
9: **return** π_{new}

GET-NEW-STATE-1()

1: $i := random(1, n)$ ▷ “Выбираем” задание

2: $p_{new} := random(1, m)$ ▷ и процессор

▷ Генерируем новый номер в очереди на выполнение

3: $\pi_{new} :=$ GENERATE-NEW-POSITION(i)

4: $from := \min\{\pi(i), \pi_{new}\}$ ▷ Запоминаем для пересчета времен

5: MOVE-TASK(i, p_{new}, π_{new}) ▷ Перемещаем задание

▷ Пересчитываем времена начала заданий, которые могли передвинуться

6: REFRESH-START-TIMES($from$)

2.4.2. Вторая функция перехода в новое состояние

До этого для генерации номера задания было взято равномерное распределение. Но можно взять распределение такое, чтобы вероятность выбора заданий с более загруженных процессоров была больше.

Назовем такую функцию GET-NEW-STATE-2().

2.4.3. Третья функция перехода в новое состояние

Теперь обратимся обратно к жадному алгоритму, с помощью которого получили начальное состояние. Этот алгоритм распределяет задания по процессорам довольно “равномерно”. Поэтому перенос одного задания на другой процессор скорее всего будет увеличивать длину расписания.

Приходим к новой идее: выбираем два задания на разных процессорах и меняем их местами: переносим каждое из них на процессор другого.

GET-NEW-STATE-3()

1: $i_1 := random(1, n)$ ▷ “Выбираем” первое задание

2: $i_2 := random(1, n)$ ▷ “Выбираем” второе задание

3: **while** $p_{i_1} = p_{i_2}$ **do**

4: $\quad i_2 := random(1, n)$

5: π^1_{new} :=GENERATE-NEW-POSITION(i_1)

6: $from := \min\{\pi(i_1), \pi^1_{new}\}$ ▷ Начинаем запоминать для пересчета времен

7: MOVE-TASK(i_1, p_{i_2}, π^1_{new}) ▷ Перемещаем первое задание

8: π^2_{new} := GENERATE-NEW-POSITION(i_2)

9: $from := \min\{from, \pi(i_2), \pi^2_{new}\}$

10: MOVE-TASK(i_2, p_{i_1}, π^2_{new}) ▷ Перемещаем второе задание

▷ Пересчитываем времена начала заданий, которые могли передвинуться

11: REFRESH-START-TIMES($from$)

Глава 3

Статистические результаты применения метода к задаче

N	Алгоритм	T_{max}	T_{min}	$\mathscr{T}(k)$	k_{max}
1	GREEDY-ALGORITHM()	—	—	—	1
2	GET-NEW-STATE-1()	$\frac{n^2}{1000}$	1	$0.999^k \cdot T_{max}$	$\log_{0.999} \frac{T_{min}}{T_{max}}$
3	GET-NEW-STATE-1()	n	1	$T_{max} - \frac{T_{max} - T_{min}}{k_{max}} \cdot k$	$7 \cdot n + 3000$
4	GET-NEW-STATE-2()	$\frac{n^2}{1000}$	1	$0.999^k \cdot T_{max}$	$\log_{0.999} \frac{T_{min}}{T_{max}}$
5	GET-NEW-STATE-2()	n	1	$T_{max} - \frac{T_{max} - T_{min}}{k_{max}} \cdot k$	$7 \cdot n + 3000$
6	GET-NEW-STATE-3()	$\frac{n^2}{1000}$	1	$0.999^k \cdot T_{max}$	$\log_{0.999} \frac{T_{min}}{T_{max}}$
7	GET-NEW-STATE-3()	n	1	$T_{max} - \frac{T_{max} - T_{min}}{k_{max}} \cdot k$	$7 \cdot n + 3000$

Таблица 3.1: Список примененных методов.

N	Средняя относительная ошибка	Среднеквадратическое отклонение относительной ошибки	Максимальная относительная ошибка	В скольки процентах случаев получено оптимальное расписание	Средняя абсолютная ошибка
1	1.57666	0.298644	2.19927	3.9%	837.263
2	1.21511	0.198583	1.80246	11.3%	309.965
3	1.13863	0.146999	1.62754	13.7%	182.337
4	1.19298	0.182492	1.84848	5.8%	218.918
5	1.10155	0.114651	1.53237	12.9%	125.604
6	1.10944	0.110456	1.55246	11.7%	138.906
7	1.06601	0.080904	1.36384	13.7%	67.702

Таблица 3.2: Статистические результаты.

Заключительные замечания

В работе были описаны, реализованы и проанализированы разные вариации метода имитации отжига применительно к общей задаче о расписании. Как и стоило ожидать лучший результат дает вариант алгоритма, где для перехода к новому расписанию два задания меняются местами, а количество итераций линейно зависит от количества заданий (этого следовало ожидать, так как жадный алгоритм, используемый для получения начального решения старается распределить задания равномерно по процессорам).

В итоге мы получили полиномиальный алгоритм, средняя ошибка которого на наших экспериментальных данных составила всего 1.06601.

Как уже говорилось во введении наша программа опубликована в сети Интернет по адресу:
`https://bitbucket.org/noscode/simulated_annealing_for_general_scheduling_problem/`.

Список алгоритмов

Литература

1. Григорьева Н.С. Теория расписаний методические указания // СПбГУ кафедра Исследования Операций 1995 г.

2. S. G. Ponnambalam, N. Jawahar, P. Aravindan A simulated annealing algorithm for job shop scheduling, Production Planning & Control: The Management of Operations, 10:8, 1999г, с. 767-777.

3. Костенко В. А. Задача построения расписания при совместном пректировании аппаратных и программных средств // Программирование, 2002г, №3, с. 46-80.

4. `http://www.kasahara.elec.waseda.ac.jp/schedule/`